IMPLICATIONS OF RESPONSIVE SPACE ON THE FLIGHT SOFTWARE ARCHITECTURE

Jonathan Wilmot, NASA GSFC, Greenbelt MD

ABSTRACT

The Responsive Space initiative has several implications for flight software that need to be addressed not only within the run-time element, but the development infrastructure and software life-cycle process elements as well. The run-time element must at a minimum support "Plug & Play", while the development and process elements need to incorporate methods to quickly generate the needed documentation, code, tests, and all of the artifacts required of flight quality software. Very rapid response times go even further, and imply little or no new software development, requiring instead, using only pre-developed and certified software modules that can be integrated and tested through automated methods. These elements have typically been addressed individually with significant benefits, but it is when they are combined that they can have the greatest impact to Responsive Space. The Flight Software Branch at NASA's Goddard Space Flight Center has been developing the run-time, infrastructure and process elements needed for rapid integration with the Core Flight software System (CFS) architecture. The CFS architecture consists of three main components; the core Flight Executive (cFE), the component catalog, and the Integrated Development Environment (IDE). This paper will discuss the design of the components, how they facilitate rapid integration, and lessons learned as the architecture is utilized for an upcoming spacecraft.

INTRODUCTION

Achieving rapid deployment of flight quality software requires a different approach than our heritage process. Using a standard spacecraft development cycle, flight software typically takes 2-3 years from requirements definition to spacecraft integration and test (I&T). Launch is typically a year later. More complex spacecraft can take even longer. To be fair, much of the work at NASA GSFC requires developing one of a kind custom science spacecraft where the software process is typically paced by the hardware schedule. But even with custom spacecraft there is a great deal of commonality that could be exploited. At Goddard the main driver for changing the development process is cost, but with software being a labor intensive process, cost and schedule can be directly related. An obvious way to reduce cost and schedule is to increase the amount of software reuse. After performing extensive analysis of heritage missions the Flight Software Branch at Goddard has adopted a product line approach where the commonality is exploited and the whole development cycle is targeted for formal reuse.

For the product line approach to have a significant impact, the software development process must reuse all artifacts of the development cycle, not just the code. This includes the process artifacts from requirements, design, test, reviews, and all of the associated documentation. A component based reuse process that addresses these aspects has been under formal development at Goddard Space Flight Center for about two years. This process and support tools, called the Core Flight System (CFS), is based on three major components, a small runtime flight executive, an expandable catalog/library of reusable software components, and a integrated development environment to bring it all together. When complete, the CFS is designed to allow components to be selected, configured, and deployed into running systems with all the process artifacts, which can reduce aspects of the development cycle from years to months.

RUNTIME EXECUTIVE

To provide an operating basis for the CFS component reuse model we have developed a small footprint core system executive that supports runtime plug-and-play of software components that conform to the core Application Programmer Interface (API). This allows software developers to select components from the catalog/library and quickly integrate them into new systems. This system executive is called the core Flight Executive or cFE. The core Flight Executive, denoted in blue the figure 1, consists of an operating system abstraction layer, a hardware abstraction layer, a set of common system services, and a Publish and Subscribe messaging middleware. This executive, while facilitating reuse, also adds a great deal of development and operational flexibility in the areas of rapid prototyping, on-orbit software maintenance, redundancy management, and communications.

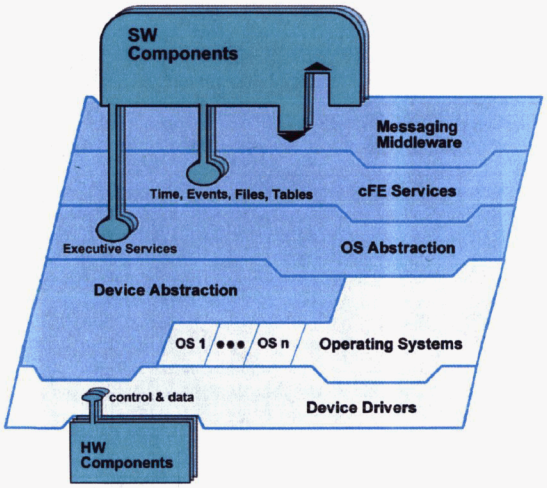


Figure 1

CORE COMPONENT SERVICES

The core Flight Executive is the minimal set of component services required for any generic processing element in the flight domain. This core is compiled and linked as a unit generating a single object and several configuration files. The core services are split into the following 7 logical subsystems: 1) Messaging Middleware, 2) Event Handler, 3) OS Abstraction Layer, 4) Executive Services, 5) File Services, 6) Table Management, 7) Time Management. Each service typically includes a library and an external interface application. The library contains the runtime services and the APIs. The external interface applications implement any required service state machines and contain the high level command and telemetry interfaces for ground system interactions. One of the subsystems, the OS Abstraction, is implemented as a library only. This subsystem may be used without the others as a stand-alone product. Prior to deployment, each service can be configured and scaled to match the mission requirements. These configuration parameters are isolated to a few files which affect system performance and resources utilization allowing the cFE to quickly target a wider range of platforms. All of the cFE services maintain information for each user component and its tasks in order to provide a way of cleaning up after an application that may have crashed and to provide information to the ground for performance and resource monitoring. This feature is especially useful during development and can be used to restart single components on-orbit in response to contingencies.

Messaging Middleware

The software message bus architecture has a long flight heritage at Goddard. It was first implemented in 1990 as the abstraction for inter-task communications. The core service Software Bus (SB) adds a dynamic Publish and Subscribe messaging interface and handles both local and distributed inter-task communications, in a transparent way. The primary abstraction of Software Bus is to provide a mechanism for message transmission such that data providers send packets without knowledge of the data consumers. This allows the one or more subscribers to be on any platform within scope of the bus and not affect the sending application. This location transparency is a critical feature for rapid integration, component portability and reconfiguration. Only device drivers need to be tied to specific hardware platforms. Figure 2 illustrates the spacecraft messaging concept. All components connect to the same bus regardless of platform.

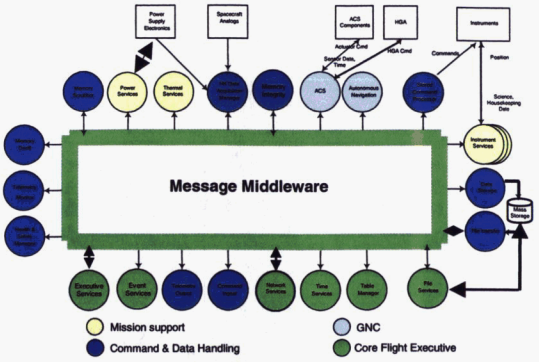


Figure 2

The Software Bus message-based subscription supports the heritage concepts of one-to-one, one-to-many, and many-to-one routing configurations along with the Poll (non-blocking) and Pend (blocking w/wo timeout) options for message receipt. The quality of service (QoS) concepts of priority and reliability are implemented for all off-processor messages.

Event handler

Event messages are informational text generated by an application in response to commands, software errors, hardware errors, application-initialization, etc. Event messages are sent to alert the operators that some significant event on- board has occurred. Event messages may also be sent for debugging application code during development, maintenance, and testing. The event handler implements the operator interface, filtering, formatting, sending, and error counters for all event messages.

OS Abstraction Layer (OSAL)

The OSAL is the API and library abstractions for common operating systems used in development and deployment of flight software systems, including embedded and desktop systems. To keep it simple, OSAL implements only the subset of operating system functions as required from the heritage analysis and system goals. The OSAL[1] is a Goddard stand-alone open source project. See reference for more information on supported operating systems and target platforms.

Executive Services

The Executive Services (ES) subsystem provides system startup, interfaces for run-time control of the cFE and component applications, system logging and interrupt/exception handling. Many of the control features for stopping, starting, and inspecting component applications were not present in the heritage architectures but are required to meet the fundamental design goals of CFS.

On system startup, the boot firmware copies the cFE and operating system from non-volatile memory into pre-determined addresses in volatile memory. Control is then transferred to the OS and then to ES which handles the startup of the cFE and the rest of the cFE Applications as indicated in a configuration file. Each application has one main task and may have additional child tasks. For embedded flight platforms, the cFE itself is linked and loaded with the RTOS and BSP as a single static executable in non-volatile memory. On desktop platforms, such as Linux, the cFE is loaded as a process.

File Services

The File Service implements the standard file access interfaces for embedded systems along with interfaces for operator control. It will also contain the APIs for retrieving the file tracking data. The tracking data allows for system monitoring and application cleanup.

Table Management

A table is a related set of data values (equivalent to a C structure or array) that can be loaded and dumped as a single unit by the ground. Tables are used in the flight code to give system operators the ability to update constants used by the flight software during spacecraft operation without the need for patching the software. Some tables are also used for dumping infrequently needed status information to the ground on command.

The cFE implements Table Services using a different paradigm than has been used in heritage missions. A Table is considered a shared memory resource. An Application requests the creation of the shared memory from the cFE and the Application must routinely request access and subsequently release access to the Table. In this way, Table Services is able to perform Table updates without the Application being involved.

Time Management

Time Management handles the ground time interface, intra-processor time distribution and provides the time utilities API for local applications. For intra-processor time

distribution, time management provides both server and client interfaces.

COMPONENT CATALOG/LIBRARY

Once given a base of standard services and the associated Application Programmer Interfaces (API), the team could focus on building a set of common spacecraft software components. The selection of common components is based on the heritage analysis which allows a significant amount of code to be ported from existing spacecraft. Even with the heritage code base, the construction of the component library has been, and still is, the most tedious element of the CFS. Modular component standards for requirements, tests, configuration management, change tracking, and other artifacts needed to be established across the software development organization. Previously individual projects established these standards with little or no coordination with other projects. It was only with the consolidation of flight software developers in one organization that the standards were adopted.

Each component in the catalog is designed for modularity only interfacing to other components through the cFE APIs. The configuration management (CM) tool is setup to supports this modularity by keeping the requirements, code, unit tests, build tests, directory & build structure and documentation for each component separate. This allows each component to be extracted and managed as a whole unit isolated from other components. Another often overlooked aspect of reuse approaches is commonality of the compile, link, and make tools. The adoption of the standard open source GNU tools facilitates commonality by removing dependencies on proprietary tools. Each component has local "make" files and associated links to a standard set of environment variables that integrate it to the system make environment. The end result of this effort is that a component can be "checked out" of the CM tool and built with no modification to the local directory structure or make utilities.

Key to the cFE is the fact that individual catalog components can be compiled and statically linked separately from the cFE and other cFE applications. Static linking means the global Load Image symbols don't change when the application is bound to the original image.

INTEGRATED DEVELOPMENT ENVIRONMENT

The Integrated Develop Environment (IDE) provides a set of tools and interfaces for mission engineers to select and configure components and deploy the system. The environment is based of the open source Eclipse[2] tools now being adopted by many commercial vendors. Although this is a work in progress, we expect to automate several aspects of the development process. Eclipse JAVA plugins are being developed for cFE parameter configuration, component selection and configuration, CM and discrepancy/change request tracking (DCR), system build, requirements generation, command and telemetry database generation, test procedures, and documentation. Figure 3 illustrates the concept. Graphical User Interface (GUI) plug-ins are used to assist the mission engineer in selecting, determining dependencies, and configuring the cFE and components, while other plug-ins generate the component artifacts.

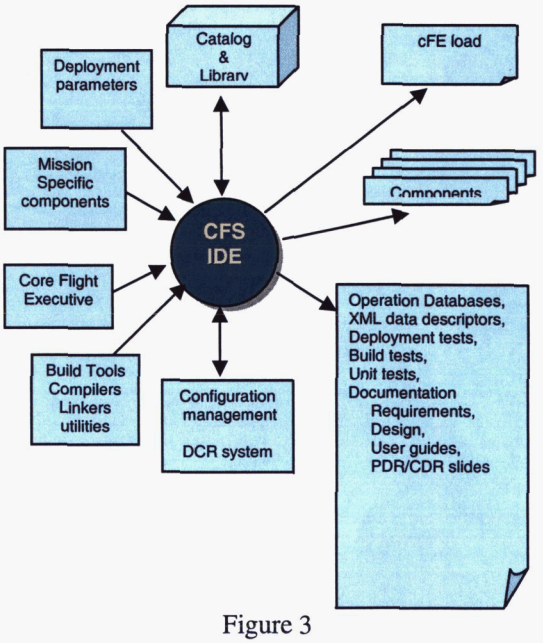


Figure 3

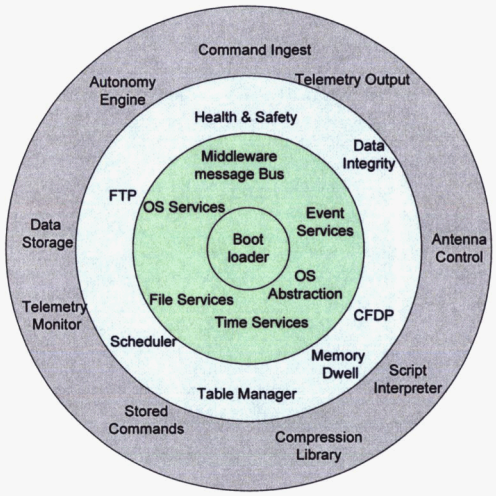
RAPID SYSTEM BUILDUP

For rapid system buildup the cFE/CFS concept provides a ready solution. Given one of the supported processing platforms, the common

components can be selected from the component library, configured for the mission, and built using the make tools. The application files can then be transferred to the target platform RAM or non-volatile memory, and started. During startup the components register for resources and subscribe to messages. Other components can subscribe to any output messages and the telemetry can be routed to the ground. New mission components can be developed from the component template and loaded using the same method. Rapid prototyping of new concepts are supported with the runtime load and unload features. Engineers can quickly make changes, assess performance and unload/reload as needed. The cFE tracks registration and resource allocations during the load and startup and allows component cleanup when needed from other applications.

CONCLUSION

The CFS can be used to rapidly build new flight systems. Using a standard set of supported avionics boxes integrated with a ring of standard cFE services, mission developers can add a system ring of catalog/library software components to create the command & data, guidance navigation & control, power, thermal, and instrument subsystems needed to assemble a complete spacecraft. Processing platforms can be chosen for power requirements, performance and other criteria based on system needs knowing the abstraction layers will isolate the reusable components from the platform choices. A prototype cFE was flown on the CHIPsat mission in December 2005. The first class B mission to use the cFE and some of the library components will launch in late 2008 or early 2009 on the Lunar Reconnaissance Orbiter (LRO).



[1] http://www.eclipse.org/

[2] http://opensource.gsfc.nasa.gov/projects/osal/osal.php